Leveraging Protocol Knowledge in Slack Matching

Girish Venkataramani
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15213
girishv@ece.cmu.edu

Seth C. Goldstein
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15213
seth@cs.cmu.edu

ABSTRACT

Stalls, due to mis-matches in communication rates, are a major performance obstacle in pipelined circuits. If the rate of data production is faster than the rate of consumption, the resulting design performs slower than when the communication rate is matched. This can be remedied by inserting pipeline buffers (to temporarily hold data), allowing the producer to proceed if the consumer is not ready to accept data. The problem of deciding which channels need these buffers (and how many) for an arbitrary communication profile is called the slack matching problem; the optimal solution to this problem has been shown to be NP-complete.

In this paper, we present a heuristic that uses knowledge of the communication protocol to explicitly model these bottlenecks, and an iterative algorithm to progressively remove these bottlenecks by inserting buffers. We apply this algorithm to asynchronous circuits, and show that it naturally handles large designs with arbitrarily cyclic and acyclic topologies, which exhibit various types of control choice. The heuristic is efficient, achieving linear time complexity in practice, and produces solutions that (a) achieve up to 60% performance speedup on large media processing kernels, and (b) can either be verified to be optimal, or the approximation margin can be bounded.

1. INTRODUCTION

A pipelined circuit is well-balanced if all computation stages are allowed to execute as soon as the input data is available. Stalls in the pipeline occur due to various reasons; one fundamental cause of pipeline stalls is a mis-match between the data production and consumption rates. Consider the example in Fig. 1a. It shows two pipelined circuit loops, C1 and C2, executing concurrently, which must synchronize twice; at stages X and Y. Assuming that the delay along cycle C1 is much greater than the delay along C2, the critical cycle of execution for this circuit is shown in Fig. 1b. This critical cycle passes not just through forward edges of the graph, but also through a backward edge (along channel AX). Such backward edges are used to acknowledge previously produced data items. Stage A cannot generate new data until such an acknowledgment is received from X.

This bottleneck occurs because of a re-convergent path that is unbalanced, Y-X $>$ (Y-A + A-X). Data items from previous iterations clog up the shorter path, thus constricting the progress of A until X consumes the waiting data items. Inserting the buffer acts as temporary storage for new data items, thus enabling A to continue along path AY; the result is a shorter critical cycle latency. In this paper, we show that backward edges along the critical cycles are indicative of the presence of bottlenecks. The backward edge is manifest in different forms and patterns depending on the design style, e.g., in asynchronous [9] and latency insensitive synchronous systems [6], it is an explicit control signal representing the acknowledgment of a handshake; for a synchronous FIFO, it may be the *fifo_full* signal. When the critical cycle of execution is free of such backward edges, then it represents the largest cycle in the circuit constructed from just forward edges. This cycle is referred to as the *algorithmic cycle* of the circuit, and has been shown to be a performance bound for a given pipelined circuit design [1, 3], and is free of communication bottlenecks.

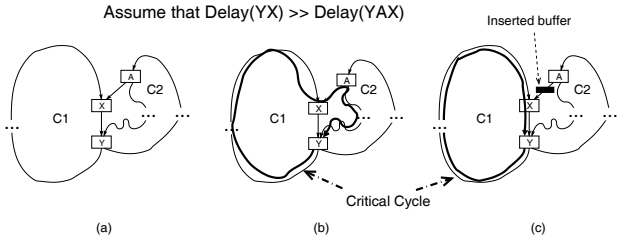

Figure 1: *(a) An example demonstrating the need for slack matching; (b) the critical cycle before, and (c) after slack matching.*

Slack matching is an optimization that determines the minimum number of buffers that must be inserted in order to guarantee that the critical cycle of execution is the algorithmic cycle. Previous approaches to slack matching [1, 8, 6] have cast the optimization as a constraint satisfaction problem that meets the algorithmic cycle time constraint. The complexity of these solutions are NP-complete. Our solution is an efficient heuristic which is inspired by examining the properties of the critical cycle, which we refer to as the Global Critical Path (GCP) [12]. We observe that: (a) it is possible to accurately model the GCP for a circuit in a given initial state; (b) the backward edges that represent communication bottlenecks will manifest themselves as a precise pattern on the GCP, whose topology is directly correlated to the protocol used to communicate over a given channel; and (c) when a buffer is inserted on such a channel, it is possible to eliminate the bottleneck, and to efficiently and accurately re-compute the GCP to reflect the new critical cycle.

Based on these observations, we propose an algorithm that starts with the GCP for an initial state of the circuit, examines it for bottleneck patterns representing mis-matched communication rates, inserts pipeline buffers to eliminate the bottlenecks, updates the GCP and looks for more opportunities. This is repeated until no more bottlenecks can be removed from the GCP. We show that this algorithm is efficient and, in practice, achieves linear time complexity in the number of communication channels in the system. The algorithm is a heuristic since we cannot prove that the algorithmic cycle time will be met, or that the minimum number of buffers will be inserted. However, experimental results indicate that the heuristic achieves a solution that is provably either optimal, or is within a 5% approxi-

ICCAD '06 November 5-9, 2006, San Jose, CA

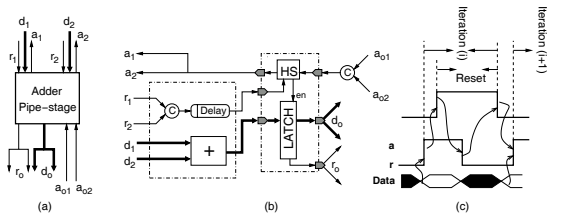

Figure 2: *(a) An asynchronous communication channel is associated with two control signals (*req *and* ack*), (b) a typical asynchronous pipeline stage, and (c) A 4-phase handshake protocol.*

mation margin. The result is significant performance improvement (up to 60% speedup) and improved energy efficiency, as reported by post-layout simulations.

In this paper, we focus on an asynchronous communication protocol, the four-phase bundled data protocol [4]. However, the concepts proposed can be adapted to other protocols as well. Section 2 introduces the modeling and analysis framework that infers the necessary metrics for use in the optimization phase. Section 3 describes the slack matching algorithm, which we compare to previous work in Section 4. In Section 5, we report experimental results, and conclude in Section 6.

2. MODELING AND ANALYSIS

Asynchronous circuit design is characterized by the absence of a global synchronization signal, e.g., a clock. Fig. 2a shows the interface of a typical asynchronous stage. Each data channel carries additional control signals, which participate in the handshake protocol. Fig. 2b shows the architecture of this stage: it has a controller, *HS* unit, which implements the local handshaking and latches the result. Fig. 2c shows a four-phase handshake protocol [4], which is commonly used in asynchronous designs for data transfers between stages; we apply our slack matching algorithm in the context of this protocol.

2.1 System Modeling

We model the performance behavior of these circuits at the pipeline stage granularity. Specifically, we capture dependence relations between the handshake events at the pipeline stage interface. An analysis phase then computes the steady-state timing dependencies between these events, which form the input to the slack matching algorithm. We formally describe this modeling and analysis phase in the remainder of this section.

The behavior of a given circuit is captured by the system, $N = (E, B, In, Out, X, R, M, M_0)$. An *event*, $e \in E$, represents a transition on a control signal in the circuit. It is *alive* when the equivalent circuit transition has fired. The set of all live events is given by M, and M_0 is the set of events initially live. A *behavior*, $b \in B$, defines how an event becomes alive. It is associated with some input events ($In : B \mapsto 2^E$), and generates some output events ($Out : B \mapsto 2^E$). It is *satisfied* when all its input events are alive ($In(b) \subseteq M$). In the absence of choice [13], a behavior can *fire* once it is satisfied, and this will make all its outputs live.

Choice and conditional flow semantics [13] are modeled by allowing multiple behaviors to fire the same events. Thus, $\exists b_1, b_2 \in B$, s.t., $Out(b_1) \cap Out(b_2) \neq \emptyset$. For unique (deterministic) choice, only one of these behaviors can be satisfied at any instant, i.e., $In(b_1) \cup In(b_2) \not\subset M$ is an invariant. Arbitration (non-deterministic) choice is modeled by specifying mutually exclusive behaviors, $X : B \mapsto 2^B$. A member ($X(b) \subseteq B$) is a set of behaviors which may be satisfied simultaneously, but their outputs are mutually exclusive. Thus, b fires iff: $\forall b' \in X(b), Out(b') \cap M = \emptyset$. If two members of $X(b)$ are satisfied in the same instant, then one of them is probabilistically chosen to fire, reflecting the non-deterministic firing semantics of arbitration behaviors. If b is a non-choice or unique choice behavior, $X(b) = \emptyset$.

Finally, we define $R : E \mapsto E$, which specifies how an event becomes *dead*; an event $e \in M$ is removed from M when $R(e)$ becomes alive. For example, the rising transition of a circuit signal will render the falling transition of the same signal as dead. The following models the adder in Fig. 2a, assuming that it implements the protocol described in Fig. 2c:

$$
\begin{array}{lll}
E = & \{r_1\uparrow, r_1\downarrow, r_2\uparrow, r_2\downarrow, a_1\uparrow, a_1\downarrow, a_2\uparrow, a_2\downarrow, & \\
 & r_o\uparrow, r_o\downarrow, a_{o_1}\uparrow, a_{o_1}\downarrow, a_{o_2}\uparrow, a_{o_2}\downarrow\} & \\
B = & \{b_1, b_2, b_3\} & X = \emptyset \\
In(b_1) = & \{r_1\uparrow, r_2\uparrow, a_{o_1}\downarrow, a_{o_2}\downarrow\}, & Out(b_1) = \{r_o\uparrow, a_1\uparrow, a_2\uparrow\} \\
In(b_2) = & \{a_{o_1}\uparrow, a_{o_2}\uparrow\}, & Out(b_2) = \{r_o\downarrow\} \\
In(b_3) = & \{r_1\downarrow, r_2\downarrow\}, & Out(b_3) = \{a_1\downarrow, a_2\downarrow\} \\
M_0 = & \{e\downarrow \mid e \in E\} & \forall e \in E,\ R(e\uparrow) = e\downarrow, R(e\downarrow) = e\uparrow
\end{array}
$$

Behavior b_1 describes how the adder functions. Once its input data channels are valid (indicated by $r_1\uparrow$ and $r_2\uparrow$), and its previous output has been consumed (i.e., $a_{o_1}\downarrow$ and $a_{o_2}\downarrow$), the adder processes its inputs, generates a new output ($r_o\uparrow$), and acknowledges its inputs ($a_1\uparrow$ and $a_2\uparrow$). Behaviors b_2 and b_3 describe the reset phase of the handshake as shown in Fig. 2c. Notice that the sets In and Out only specify the input and output handshake signals of the pipeline stage interface. Thus, behaviors in the model are restricted to control events (handshake events, in particular), implying that internal events, implementation details, and datapath logic are abstracted away. This not only leads to a large reduction in the model size, but also decouples system modeling from system implementation.

2.2 Model Analysis

Performance can be analyzed by associating a delay, $D : B \mapsto \mathbb{R}$, with every behavior in the model. This is the expected execution latency through the micro-architectural blocks that b describes. Depending on the system and the environment, these delays may be deterministic or stochastic. The model can now be analyzed by simulating the firing sequence of events and behaviors — when $b \in B$ is satisfied, it fires after $D(b)$ time units, generating its output events, which in turn satisfy other behaviors. This analysis is similar to techniques proposed for analyzing event-rule systems [3], marked graphs [7], and Petri-Nets [13]. The weaknesses of all these techniques are (a) their inability to scale when analyzing large problem sizes (on the order of thousands of events and beyond), and (b) their inability in handling systems with choice.

To overcome these difficulties, we employ a trace-based simulation technique to focus the analysis on the most commonly expected input vectors. Since there is a one-to-one correspondence between events and signal transitions, and between behaviors in the model and micro-architectural blocks in the circuit, we simulate the synthesized and laid out circuit using commercial simulators (like Modelsim), and analyze the model during circuit simulation. This results in an accurate delay model, and our experiments reveal a low overhead (about 6%) in analyzing the model during simulation.

Slack. The analysis produces slack relations for every $b \in B$; if in the k^{th} firing, inputs of b (where $|In(b)| = m$) arrived at times, $\{t_1 \leq \cdots \leq t_i \leq \cdots \leq t_m\}$, then slack on event, e_i is given by $Slack^k(e_i, b) = (t_m - t_i)$. This implies that e_i arrived $Slack^k(e_i, b)$ time units before the last arriving event, e_m, which satisfied b. Event, e_m, with the least slack is referred to as the *locally critical* event, $Crit(b)$.

Global Critical Path (GCP). If b_{last} is the last behavior to fire in the timed execution, then we define the GCP as the sequence of behaviors, $\langle b_1, \ldots, b_i, \ldots, b_{last} \rangle$. For any two consecutive (b_i, b_{i+1}) in the GCP, $Crit(b_{i+1}) \in Out(b_i)$. Equivalently, it is also the event sequence, $< e_1, ..., e_{last} >$, s.t., $e_i = Crit(b_i)$.

The slack and GCP metrics thus constructed are described for a timed execution [13]. Thus, there may be multiple instances of a given b in the GCP. Since the number of times a behavior fires can be potentially large, we summarize slack and GCP on an untimed model. For a given b that fires N_b times, slack on $e_i \in In(b)$ is given by $Slack(e_i, b) = (\sum_{j=1}^{N_b} Slack^j(e_i, b))/N_b$. Summarizing GCP and slack

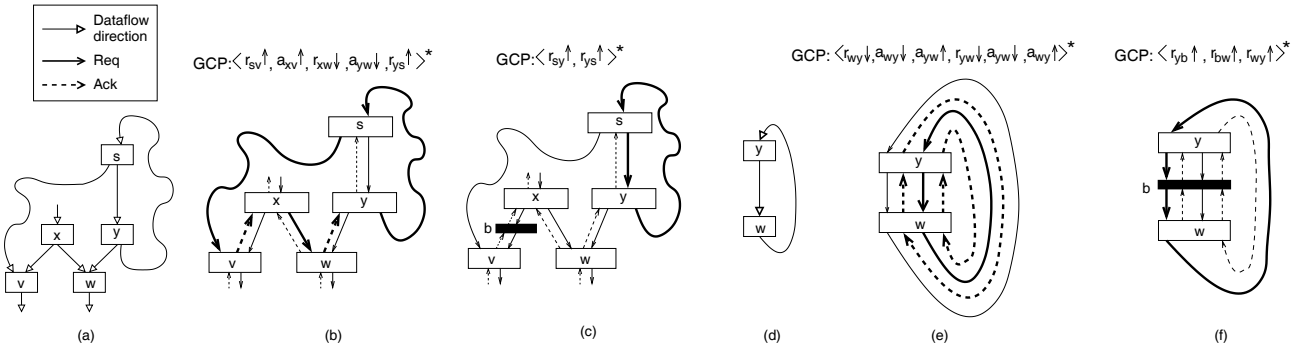

Figure 3: *Two causes of bottlenecks: re-convergent paths (a-c) and short loops (d-f). For each, the figure shows system topology with data dependencies (a,d), the GCP in the presence of a bottleneck (b,d), and the GCP after buffer insertion (c,f). Bold edges mark the GCP.*

in this manner leads to information loss, but it also biases the analysis for the common case. These metrics are key components in our slack matching algorithm, which is described in the following section.

3. SLACK MATCHING

Beerel [1] notes that there are only two causes for communication bottlenecks: *re-convergent paths* and *short loops*. For example, the design in Fig. 1 has a re-convergent path that causes a bottleneck. The key insight to finding the bottlenecks in both these cases is the observation that these bottlenecks are manifest as a precisely defined pattern (critical event sequence) on the GCP. The shape of this sequence is defined by the communication protocol employed.

Re-convergent paths. These are the most common source of bottlenecks in large concurrent systems. To find and eliminate these bottlenecks, an analysis of the communication protocol is essential. In some cases, the re-convergent path is not immediately apparent, but can be uncovered only by examining the protocol interactions. For example, there are apparently no re-convergent paths in Fig. 3a, but when the handshake event interaction is observed, we notice that a re-convergent path exists, forking at stage s. The join point, however, is not a single stage, but three stages together (v, x and w). To understand why, consider Fig. 3b which shows for each channel, the *Req*/*Ack* signals that implement the 4-phase handshake described in Fig. 2c. The model describing the firing behavior for these signals is similar to the adder example in Section 2.1. Assume that the path sv is much longer than sy. If r_{mn} and a_{mn} are the *Req*/*Ack* signals of the channel from m to n, then $r_{sy}\uparrow$ arrives much earlier than $r_{sv}\uparrow$. Stage y can begin the handshake on channels ys and yw, firing $r_{ys}\uparrow$ and $r_{yw}\uparrow$. After this, w acknowledges with $a_{yw}\uparrow$, and the reset phase begins when y generates $r_{yw}\downarrow$. However, the reset phase cannot complete since w must wait for $r_{xw}\downarrow$ from x, before generating $a_{yw}\downarrow$. Event $r_{xw}\downarrow$, in turn, is awaiting $a_{xv}\uparrow$, which in turn is awaiting $r_{sv}\uparrow$, the latter representing data along the longer path from s. Thus, stages v, x and w are all involved in the handshake along both paths, sw and sv, and together they form the join point of the re-convergent path from s.

If the cycle $s-y-s$ is shorter than the path sv, then, in steady state, $r_{sy}\uparrow$ always arrives earlier than (and is waiting for) $a_{yw}\downarrow$, i.e., the next iteration is throttled due to the late completion of the handshake reset phase of the previous iteration, resulting in a bottleneck; the GCP is: $\langle r_{sv}\uparrow, a_{xv}\uparrow, r_{xw}\downarrow, a_{yw}\downarrow, r_{ys}\uparrow\rangle^*$. In other words, if no bottleneck exists, then a stage should process its inputs as soon as they all arrive. In essence, the appearance of the reset event sequence, $\langle a_{xv}\uparrow, r_{xw}\downarrow, a_{yw}\downarrow\rangle$ on the GCP is indicative of a slow consumer, resulting in a bottleneck.

If a buffer is inserted on channel xv (Fig. 3c), then the late ack, $a_{xv}\uparrow$ can now be generated early since it no longer needs to wait for $r_{sv}\uparrow$; in other words, the buffer acts as temporary storage for the early arriving input at stage v. The ack event can now be generated as soon as $r_{xv}\uparrow$ fires, implying that inserting a buffer accelerates the generation of $a_{xv}\uparrow$ by the amount of time $r_{xv}\uparrow$ was kept waiting, which is nothing but $Slack(r_{xv}\uparrow, b')$, where b' fires $a_{xv}\uparrow$. This is a key insight in the bottleneck elimination algorithm as described in Section 3.1. The early ack event, $a_{xv}\uparrow$, in turn, produces early $r_{xw}\downarrow$ and $a_{yw}\downarrow$ events, thus eliminating the bottleneck. The GCP of the design now shifts to the topology shown in Fig. 3c.

```
Let Ignore = ∅
while(true) begin
    seq= the first p_s sequence in the GCP, that is not in Ignore
    if (seq is found) begin
        success = eliminate seq by inserting buffers (Section 3.1)
        if (success)
            update model, slack, and GCP (Section 3.2); Ignore = ∅
        else
            add seq to Ignore
    endif
    else STOP
endwhile
```

Figure 4: *The overall slack matching algorithm.*

Short Loops. A similar situation occurs in the presence of short loops as shown in Fig. 3d-f. In this case, the forward delay to transmit data around the loop (i.e., along path $[r_{yw}\uparrow \rightarrow r_{wy}\uparrow]$) is shorter than the delay to reset the handshake backwards along the same loop, resulting in the GCP shown in Fig. 3e. Again, the insertion of a buffer hastens the generation of the critical events resulting in a GCP devoid of the handshake reset events. Notice that after buffer insertion, the GCP is shorter in both cases (Fig. 3c,f).

In summary, we note that for any communication protocol, we can identify two event sequences – p_d signifies the generation of new data by the producer, and p_s is a synchronization path, which signifies the acceptance of this data by the consumer. For example, in the four phase protocol, $p_d = \langle r\uparrow\rangle$ and $p_s = \langle a\uparrow, r\downarrow, a\downarrow\rangle$; for a synchronous FIFO with protocol signals, $\{enq, deq, full, empty\}$, $p_d = \langle enq, \overline{empty}\rangle$, and $p_s = \langle deq, \overline{full}\rangle$. Ideally, we would like p_s to occur in parallel, before the next iteration's data shows up. If not, we will have new data, which we are not ready to process, implying a bottleneck. It can be shown that for any protocol, the GCP can be captured by the regular expresion: $\langle p_d^*, p_s^*\rangle^*$. In fact, the short-loop bottlenecks will always feature at least two consecutive p_s sequences on the GCP, or as in the pathological example in Fig. 3e, the GCP may be just $\langle p_s\rangle^*$. The GCP for a bottleneck-free system, however, will be $\langle p_d\rangle^*$.

Fig. 4 describes the overall algorithm for eliminating p_s sequences from the GCP. It iteratively finds a p_s sequence, inserts buffers and updates the system model. The p_s sequences are considered for elimination in topological-sort order of the GCP. This is because it is highly likely that such sequences are the cause for the appearance of others downstream. The remainder of this section describes how we eliminate p_s sequences from the GCP, and how the GCP and system model are updated once a buffer is inserted. We end the section with an analysis of the complexity and optimality of the algorithm. Although the algorithm is described for a 4-phase asynchronous handshake proto-

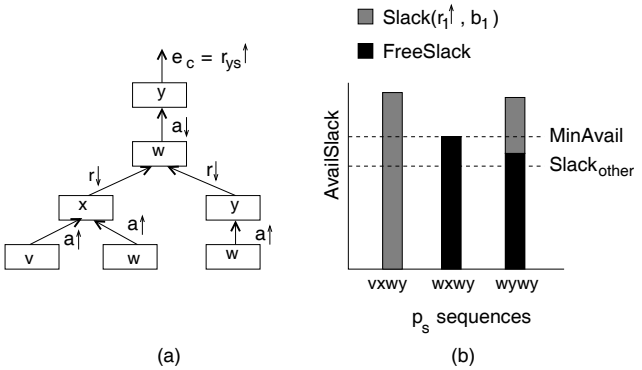

Figure 5: *A visual of (a) all the p_s paths that can precede r_{ys} in Fig. 3b (ignoring the p_s paths along ys for simplicity), and (b) computation of various slack values.*

col, it can be extended for other protocols by appropriately defining the model and the p_s and p_d event sequences.

3.1 Bottleneck Elimination

Observe that the p_s sequence $\langle a_{xv}\uparrow, r_{xw}\downarrow, a_{yw}\downarrow\rangle$ is on the GCP in Fig. 3b, because $r_{sy}\uparrow$ arrives earlier at the behavior, call it b_c, which generates $r_{ys}\uparrow$. To make the p_s sequence non-critical, we must generate it sooner by at least $Slack(r_{sy}\uparrow, b_c)$, which is the slack on the waiting event. In general, let e_c (fired by b_c) be the event that succeeds the p_s sequence on the GCP, and $Slack_{other} = Slack(e_c, b_c)$ be the slack on the waiting event at b_c that p_s must offset.

Since the p_s sequence on the GCP is a chain of locally critical events, firing the first event of p_s (call it e_1 generated by behavior b_1) earlier, will accelerate the firing of all events in p_s. This may be achieved by inserting a buffer along the channel, $chan(e_1)$, associated with e_1; a necessary condition for this, however, is that the p_d event of $chan(e_1)$, which is an input to b_1 must have positive, non-zero slack. For example, in Fig. 3b, e_1 is $a_{xv}\uparrow$; inserting a buffer along xv can hasten the generation of $a_{xv}\uparrow$ only if $r_{xv}\uparrow$ (which is the p_d event of xv) has positive, non-zero slack. Inserting a buffer ensures that the locally critical event which was constricting p_s would not affect its firing anymore, e.g., $r_{sv}\uparrow$ is not necessary to fire $a_{xb}\uparrow$ in Fig. 3c. Further, if this slack is larger than $Slack_{other}$, then we would have reached our goal. Thus, to yield a GCP free of p_s sequences, we must ensure that this condition can be achieved along every p_s sequence that could potentially precede e_c, and appear on the GCP.

Let the set P_c represent every p_s sequence that could precede a given e_c on the GCP. We can visualize this set as a tree, whose root is e_c, and every path from leaf to root represents a p_s sequence in P_c. The depth of the tree is $(|p_s|+1)$, which is four in our protocol. For example, Fig. 5a shows this tree for $e_c = r_{ys}\uparrow$ in Fig. 3b. To determine if there is sufficient slack to offset $Slack_{other}$ at b_c (which fires e_c), we compute the following:

$$
\begin{array}{ll}
FreeSlack(p_s) = & Slack(e_{|p_s|}, b_c) + \sum_{i=2}^{|p_s|} Slack(e_{i-1}, b_i) \\
AvailSlack(p_s) = & FreeSlack(p_s) + Slack(r_1\uparrow, b_1) \\
MinAvail = & \text{MIN } (AvailSlack(p_s)), \forall p_s \in P_c \\
 & \\
MinAvail & > Slack_{other} \quad (1) \\
MIN(MinAvail, Slack_{other}) & > FreeSlack(p_s) \quad (2)
\end{array}
$$

where e_i is the i^{th} event in a given p_s sequence (thus $e_{|p_s|}$ is an $a\downarrow$ event, e.g., $a_{yw}\downarrow$ in Fig. 3b), behavior b_i fires e_i, $r_1\uparrow$ is the p_d event of $chan(e_1)$, e.g., $r_{xv}\uparrow$ in Fig. 3b. This computation is illustrated in Fig. 5b. $FreeSlack(p_s)$ specifies, for each path through the tree (see Fig. 5a), the amount slack the p_s sequence has since the firing of its first event, e_1, until the firing of e_c. There exists at least one $p_s \in P_c$, i.e., one path through the tree, whose $FreeSlack(p_s)$ value is zero; this is the sequence on the GCP, e.g., *vxwy* in Fig. 5b.

$AvailSlack(p_s)$ specifies the slack that would be available if we were to insert a buffer on $chan(e_1)$, and apportion the slack from the waiting $r\uparrow$ event of $chan(e_1)$. This is equivalent to the slack on the $r\uparrow$ event of the channel between a leaf of the tree and its parent, e.g., slack of $r_{xv}\uparrow$. *MinAvail*, the minimum of all *AvailSlack*, specifies the maximum slack that can be generated by inserting buffers on the leaves of the tree. If this quantity is larger than $Slack_{other}$, then some event that is not in any $p_s \in P_c$ (e.g., $r_{sy}\uparrow$ in Fig. 3b), will become locally critical thus eliminating p_s from the GCP. Thus, (1) represents a necessary and sufficient condition for obtaining a GCP free of p_s sequences.

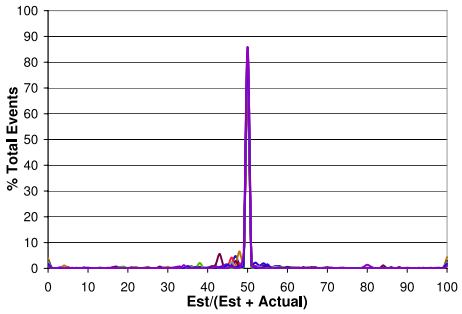

Figure 6: *The error distribution of estimating slack. If S_e and S_r are the estimated and real slack for an event, then the x-axis shows $\frac{S_e}{S_e+S_r} \times 100$; for each error value, the graph shows the percentage of total events with this error value.*

For a given p_s path in the tree, if $FreeSlack(p_s)$ is already larger than $Slack_{other}$, then no buffer is required. If, on the other hand, $FreeSlack(p_s) < Slack_{other} \leq AvailSlack(p_s)$, then we must insert a buffer on the leaf of the path, in order to leverage *AvailSlack*. We argue that even if $FreeSlack(p_s) < AvailSlack(p_s) < Slack_{other}$, we should still insert a buffer; although this buffer will not help eliminate p_s from the GCP, it will still help in accelerating its firing by $(AvailSlack(p_s) - FreeSlack(p_s))$, thus shortening the length of the GCP. Both these conditions together are captured by (2), which if satisfied results in a buffer insertion. Being conservative, however, a buffer is inserted only if this difference is larger than the latency through the buffer itself; otherwise, the output, $r\uparrow$, of the buffer will be locally critical downstream, but not necessarily on the GCP.

3.2 Metric Update

The above algorithm described how to eliminate one p_s pattern from the GCP; there may however be many more, and eliminating one may introduce others. Before looking for more opportunities, however, the model and the slack/GCP metrics must be updated to reflect changes due to the newly inserted buffer. For example, insertion of buffer, b, along channel xv in Fig. 3c, eliminates an existing channel, xv, and introduces two new channels, xb and bv, resulting in a corresponding change to the event set of the model. The slack values for the new events can be trivially determined. For example, the slack on the $r_{bv}\uparrow$ is the slack on the event it replaced, $r_{xv}\uparrow$, minus the delay of generating $r_{bv}\uparrow$ from b. Changes to slack on the other local events can be similarly determined.

Next, these changes are propagated globally. This is done by computing the minimum slack at each $b \in B$, and if this value has changed, then the difference is propagated to b', iff: $Out(b) \cap In(b') \neq \emptyset$. The effect of slack change needs to be propagated only once throughout the model, i.e., the propagation stops if the same input at the same behavior is reached again. This is because the initial slack from the analysis is a steady state property (in fact, an asymptote in systems without choice). A local change to slack implies a change to the steady state value, whose effect needs to be propagated to other events in the system.

Clearly, the slack and GCP updates are heuristics due to the lack of iteration history. We validated our algorithms by comparing the slack estimated by this update technique against a complete re-analysis of the model. The result of the comparison is shown in Fig. 6. If es-

timated slack is S_e and actual slack is S_r, then the X-axis shows the error value as $Err = \frac{S_e}{S_e+S_r} \times 100$; i.e., $Err = 50$ indicates an accurate estimation. Each trendline represents a benchmark from Table 1, and the plot shows the percentage of all events with a given error value. These results indicate that, on average, our slack estimation is accurate for more than 75% of all events, and more than 90% of events are within a $\pm 10\%$ error margin, validating this update method. The reasons for the inaccuracies are two-fold: (1) lack of iteration history, and (2) some latency estimates ($D(b)$) may not match those from actual post-layout values.

3.3 Complexity and Optimality

The computational complexity of our algorithm (as described in Fig. 4) is summarized as follows: (1) the complexity of the initial trace-based model analysis and slack/GCP construction is $O(|E|k)$, where k, the number of iterations simulated, depends on the analyzed design; (2) finding a p_s pattern on the GCP has $O(|GCP|)$ time complexity, i.e., $O(|E|)$ in the worse-case; (3) eliminating p_s depends on the fan-out and fan-in of the channels in the pattern. If f_o and f_i are the maximum fan-out and and fan-in for any behavior in the model, then $O(f_o^2 f_i)$ is the worst-case complexity; on average, f_o and f_i are about two or three; (3) updating slack and GCP each have $O(|E|)$ worst-case complexity, while updating the model occurs in constant time; (4) finally, we must bound the number of iterations the algorithm needs for convergence. In the worst-case (one buffer insertion per iteration): $\sum_i \frac{(C_i - C_o)}{D_b}, \forall C_i > C_o$, where C_o is the largest (algorithmic) cycle composed of only $r\uparrow$ events, and D_b is the delay through a pipeline buffer. That is, in the worst-case, we would have to insert buffers on all mismatched cycles. Thus, the number of iterations has $O(|E|)$ time complexity. In practice, as reported in Table 1, the algorithm converges within five iterations in most cases. The overall complexity is $O(|E|k) + O(|E|^2 f_o^2 f_i)$ in the worst-case, and $O(|E|)$ in practice, i.e., the heuristic scales linearly with the number of handshake communication channels in the model.

Finally, we address the question of optimality. There are two axes of optimality in slack matching: (a) the critical cycle must be the algorithmic cycle time, and (b) the number of pipeline buffers inserted to achieve this constraint is minimized. If the GCP, which is the critical cycle, is made up of only p_d events (which are $r\uparrow$ in the 4-phase protocol) and does not contain any newly introduced pipeline buffers, then it is guaranteed to be the algorithmic cycle. Observe, however, that in the case of short loops (Fig. 3f), buffers are always inserted on the GCP; thus the presence of buffers on the GCP does not imply non-optimality, but their absence guarantees optimality. The second optimality property, however, cannot be proven without an exhaustive analysis akin to [1]. While we cannot prove whether or not the heuristic algorithm can find an optimal solution, we can determine an approximation margin, ε, which specifies how far off the optimal a generated solution could be, in the worst case. If the GCP consists of only p_d events and does not contain any newly inserted buffers, then $\varepsilon = 0$, and the solution is optimal. If the GCP contains N_B new buffers, the delay through the buffer for generating p_d is $D(buff)$, and the algorithmic cycle time constraint is C_o, then $\varepsilon = \frac{N_B \times D(buff)}{C_o}$, in the worst case. We note that since N_B is typically very small (see Table 1), ε is also very small ($< 5\%$ in our experiments).

4. RELATED WORK

Proposals for slack matching a circuit design can be grouped into two categories: delay insertion to optimize the clock cycle time in synchronous circuits [2, 10], and those that minimize the global cycle time in data driven systems, e.g.,Latency Insensitive synchronous (LIS) designs [6] and asynchronous circuit designs [1, 8]. All these proposals take a similar approach to the solution: a target timing constraint is defined, and equations for circuit timing dependencies are set up to meet the given timing constraint, which are then solved by a linear programming optimization.

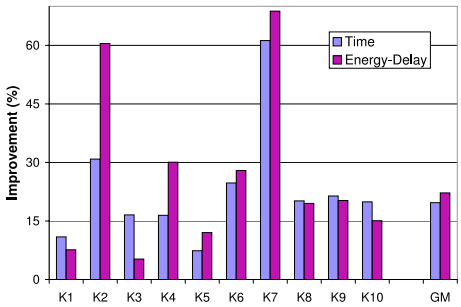

Figure 7: *Performance and energy-delay improvements after slack matching. The benchmarks are numbered in order of their listing in Table 1.*

The latter category [1, 8, 6], which attacks a problem similar to the one in this paper, defines a constraint satisfaction problem, where one of the constraints is for the critical cycle latency to be equal to the algorithmic cycle latency; the optimization objective is to minimize the number of buffers inserted to achieve this constraint. In contrast, our algorithm finds and eliminates bottlenecks by explicitly modeling them in the GCP. In terms of complexity, previous algorithms generally involve finding the cycle time constraint (the algorithmic cycle), generating a system of linear constraints, and solving the optimization problem. Determining the cycle time constraint in their approaches, and finding the GCP and slack metrics in ours, has comparable complexities. From here, we diverge from previous work. Determining the linear constraints and solving the optimization problem has been shown to be $O(m^2 n^2)$ [8] and NP-complete respectively (where m and n are the number of edges and nodes in the graph). We decompose the hard problem into smaller sub-problems that are iteratively solved. The overall complexity is $O(m^2 f_o^2 f_i)$ in the worst-case, and $O(m)$ in practice. In terms of capabilities, previous work cannot generate the LP constraints in the presence of choice. Our heuristic, on the other hand, naturally adapts to systems with all types of choice. The drawback of our approach is that creating smaller sub-problems prevents us from being able to prove its optimality. However, our results indicate that the solutions for many benchmarks are indeed optimal in the algorithmic cycle time.

5. EXPERIMENTAL RESULTS

We have implemented our algorithm in a toolflow that automatically synthesizes asynchronous circuits implementing a four-phase, bundled data protocol [11]. Synthesis uses a standard-cell flow, and targets a [180nm/2V] technology library. A property of our algorithm is that when it insert buffers, it is guaranteed to improve performance as described in Section 3.1. But, when no opportunities exist (i.e., there are no p_s sequences on the GCP), it leaves the design unchanged. In this section, we report on the results of the kernels from the Mediabench suite [5] that could be improved. We randomly picked kernels from these benchmarks, and noticed that more than 50% of those tested could be improved. Table 1 shows these kernels and their sizes in terms of the number of pipeline stages, behaviors and events.

The last column in Table 1 lists the number of iterations the algorithm took to converge, and the number of pipeline buffers inserted in each iteration. More than one pipeline buffer may be inserted per iteration as described in the slack tree analysis algorithm (see Fig. 5). All the generated solutions result in a GCP consisting of only p_d events. For seven of these kernels, the GCP does not contain any new buffers, and thus the solution is optimal ($\varepsilon = 0$). For the remainder (K3, K7 and K10), we have determined that $\varepsilon < 5\%$, in the worst case.

Our experiments indicate that the number of buffers inserted correlates with what caused the p_s bottleneck to appear on the GCP. In re-convergent paths, for which the critical event after the p_s sequence

Id	Bench	Kernel	Pipe Stages	Bhvs	Evts	Buffs per iteration
K1	adpcm_d	adpcm_decoder	191	761	1218	(5)
K2	adpcm_e	adpcm_coder	230	912	1493	(5)
K3	gsm_d	LARp_to_rp	96	341	575	(1,2,1,1)
K4	gsm_d	Short_term_ synthesis_filtering	144	582	865	(2)
K5	gsm_e	Long_term_ analysis_filtering	339	1434	2061	(1)
K6	gsm_e	Coefficients_27_39	81	312	469	(1)
K7	gsm_e	Short_term_ analysis_filtering	137	552	816	(3,2)
K8	mpeg2_d	form_component_ prediction	960	4197	6114	(1)
K9	mpeg2_e	pred_comp	925	4068	5924	(1)
K10	Huffman	HammingBitwise	131	376	570	$(1,..,1)^{22}$

Table 1: *List of kernels used. The last column shows the number of buffers inserted in each iteration of the algorithm. The Huffman circuit went through 22 iterations, with one buffer inserted in each.*

is an $r\uparrow$ (see Fig. 3b), the shorter paths of the re-convergent join need only as many buffers as the number of waves of computation that could collide at the forking point. In short loops, for which the critical event after p_s is an $a\uparrow$ event (see Fig. 3e), the waves of computation between successive iterations interfere, and these typically require more buffers.

In larger designs, re-convergent paths are dominant, and thus just a few buffers are sufficient for slack matching. In smaller designs, both types of bottlenecks may be present; moreover, fixing a re-convergent bottleneck typically gives rise to short-loop bottlenecks. Thus, more buffers are required in these cases. Acyclic designs can also exhibit bottlenecks if they are incorporated in streaming applications. For the acyclic benchmark, we injected the successive waves of inputs into the circuit, as soon as the input channels could accept them; thus we are testing its maximum capacity. Since only relative timing is measured by slack, it naturally adapts to capturing the behavior of these designs. Since we have loaded the circuit to maximum capacity, there are several waves of computation following each other in quick succession leading to the short-loop bottleneck variety. Consequently, this design required the most buffers (22), and the algorithm inserted one in each iteration of its optimization loop.

Fig. 7 shows the relative improvements in overall kernel performance and energy-delay product after applying the algorithm; a higher bar implies larger improvements. Performance refers to end-to-end latency except for the acyclic circuit (K10), for which it refers to throughput. Overall, the algorithm is capable of significantly boosting performance, over 60% in the `gsm_e` kernel, by inserting just a small number of buffers and running for a handful of iterations. Power increases due to the extra buffers, and also because the same amount of switching activity now occurs in a shorter time period. The overall effect, captured by the energy-delay metric, however, shows encouraging improvements. The geometric means of performance speedup and energy-delay improvements are 22% and 28% respectively for these experiments. The running time of the algorithm is less than ten seconds for all the designs listed in the table, and even lesser when no bottleneck exists. Comparatively, in previous work, the run-time to solve the LP formulation for large designs (thousands of events) take on the order of hundreds [1] or even thousands [6] of seconds in solving the optimization; in terms of the number of buffers inserted relative to the problem size, our algorithm is comparable to those reported in [6, 1].

6. CONCLUSIONS

We have described a heuristic slack matching algorithm that eliminates pipeline bottlenecks caused by mis-matched communication rates. The algorithm models system in terms of event slack and the Global Critical Path (GCP). We show that examining the communication protocol alone allows one to infer patterns, whose presence on the GCP indicate bottlenecks due to mis-matched communication rates. Based on these observations, the algorithm identifies bottlenecks on the GCP, eliminates them by inserting pipeline buffers, updates the system state and repeats these steps until no more bottlenecks are found. In this paper, we have focused on asynchronous circuit designs, however, the proposed modeling abstraction enables portability to other design styles as well. Further, since we rely on slack and GCP metrics, the algorithm can be applied to arbitrary graph topologies and systems with choice.

Compared to the optimal solution to slack matching, which is NP-complete, the proposed heuristic has a worst-case quadratic time complexity in the number of communication channels, and a linear time complexity, in practice. While we cannot prove whether the algorithm can converge on an optimal solution, we can bound the approximation margin for a given a solution; our experiments reveal that, in most cases, the solutions can be verified to be optimal; if not, it is within a 5% approximation margin, in the worst case. The results also show encouraging improvements (upto 60%) in performance and energy-delay.

7. ACKNOWLEDGMENTS

This research has been funded by NSF ITR Scalable Molecular Electronics (contract number CCR0205523). We would like to thank Peter Beerel, Tiberiu Chelcea, Timothy Callahan and Mihai Budiu for providing invaluable insights and engaging in discussions that helped refine the ideas presented in this paper. We also thank Shivakumar Vishwanathan and Srinath Avanadhula for providing valuable feedback that vastly improved the presentation of these ideas.

8. REFERENCES

[1] P. Beerel, M. Davies, et al. Slack matching asynchronous designs. In *ASYNC*, pp. 30–39, March 2006.

[2] E. Bozorgzadeh, S. Ghiasi, et al. Optimal integer delay budgeting on directed acyclic graphs. In *DAC*, pp. 920–925, 2003.

[3] S. Burns. *Performance analysis and optimization of asynchronous circuits*. PhD thesis, University of Utah, 1991.

[4] S. Furber and P. Day. Four-phase micropipeline latch control circuits. *IEEE TVLSI*, 4-2:247–253, 1996.

[5] C. Lee, M. Potkonjak, et al. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *MICRO*, pp. 330–335, 1997.

[6] R. Lu and C.-K. Koh. Performance optimization of latency insensitive systems through buffer queue sizing of communication channels. In *ICCAD*, pp. 227–231, 2003.

[7] C. D. Nielsen and M. Kishinevsky. Performance analysis based on timing simulation. In *DAC*, pp. 70–76, 1994.

[8] P. Prakash and A. Martin. Slack matching quasi delay-insensitive circuits. In *ASYNC*, pp. 30–39, March 2006.

[9] I. Sutherland. Micropipelines: Turing award lecture. *Communications of the ACM*, 32 (6):720–738, June 1989.

[10] B. Taskin and I. Kourtev. Delay insertion method in clock skew scheduling. *IEEE TCAD*, 25–4:651–663, 2006.

[11] G. Venkataramani, M. Budiu, et al. C to asynchronous dataflow circuits: An end-to-end toolflow. In *IWLS*, June 2004.

[12] G. Venkataramani, T. Chelcea, et al. Modeling the global critical path in concurrent systems. Technical Report CMU-CS-06-144, Carnegie Mellon University, August 2006.

[13] A. Xie, S. Kim, et al. Bounding average time separations of events in stochastic timed Petri nets with choice. In *ASYNC*, pp. 94–107, April 1999.